RX 開発環境移行ガイド

SH から RX への移行 （統合開発環境編）
(High-performance Embedded Workshop→CS+)

RJJ06J0097-0200
Rev.2.00
2017.04.20

要旨

本ドキュメントでは、SuperH ファミリ用に作成したサンプルプロジェクトを RX ファミリ用へ移行する手順についての説明を行います。

目次

1. はじめに

本ドキュメントでは、SuperH ファミリ用に作成したプロジェクトを RX ファミリ用へ移行する際に注意すべき点と、実際にサンプルワークスペースを利用した移行方法の説明を行います。

尚、本ドキュメントで用いるオプション等のバージョン依存の情報は、SuperH ファミリ用 C/C++コンパイラ V.9.04、RX ファミリ用コンパイラ V.2.06 時点の情報に基づいています。

1.1 C 言語の処理系依存

処理系依存とは、特定のハードウェアやコンパイラに依存して振る舞いが異なり、互換性のない部分を意味します。

C 言語仕様ではコードの振る舞いを各処理系で決めてよい部分があり、SuperH ファミリ用 C/C++コンパイラと RX ファミリ用 C/C++コンパイラでも処理系依存の仕様内容が異なる部分があります。

このため、同じ C 言語のソースプログラムであっても、RX ファミリ用 C/C++コンパイラのオプションを適切に設定して、処理系依存の差異に対して適切に対処する必要があります。

2. 移行の際に注意すべき機能

SuperH ファミリ用と RX ファミリ用のコンパイラは、デフォルトのオプションでは処理系依存の仕様で異なる部分が存在します。このようなオプションは、仕様を揃えるよう明示的に指定する必要があります。ここでは、SuperH ファミリ用から RX ファミリ用への移行に際し、このように注意が必要なオプションや、ソースプログラム記述について説明します。

2.1 オプション

本章では、RX ファミリ用移行時に注意が必要なオプションについて説明します。以下にその一覧を示します。

表　オプション一覧

No	機能	SuperH オプション	RX オプション	参照
1	char 型の符号指定	-	signed_char	2.1.1
2	enum のサイズ指定	auto_enum	auto_enum	2.1.2
3	double 型のサイズ指定	double=float	dbl_size	2.1.3
4	エンディアン指定	endian	endian	2.1.4
5	ビットフィールドメンバの符号指定		signed_bitfield	2.1.5
6	ビットフィールドメンバの割り付け順序指定	bit_order	bit_order	2.1.6
7	構造体の割り付け指定	pack	pack	2.1.7

2.1.1　char 型の符号指定

SuperH ファミリ用コンパイラでは符号指定のない char 型は、符号ありの signed char 型として扱います。対して RX ファミリ用コンパイラでは、デフォルトでは符号なしの unsigned char 型として扱います。

char 型が符号ありの signed char 型であることを前提に作成した SuperH ファミリ用のソースプログラムを RX ファミリ用に移行する際は、RX ファミリ用コンパイラでは"signed_char"オプションを指定します。

【書式】

signed_char

unsigned_char　　：デフォルトは unsigned_char

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの"共通オプション"タブ内で次のように設定します。

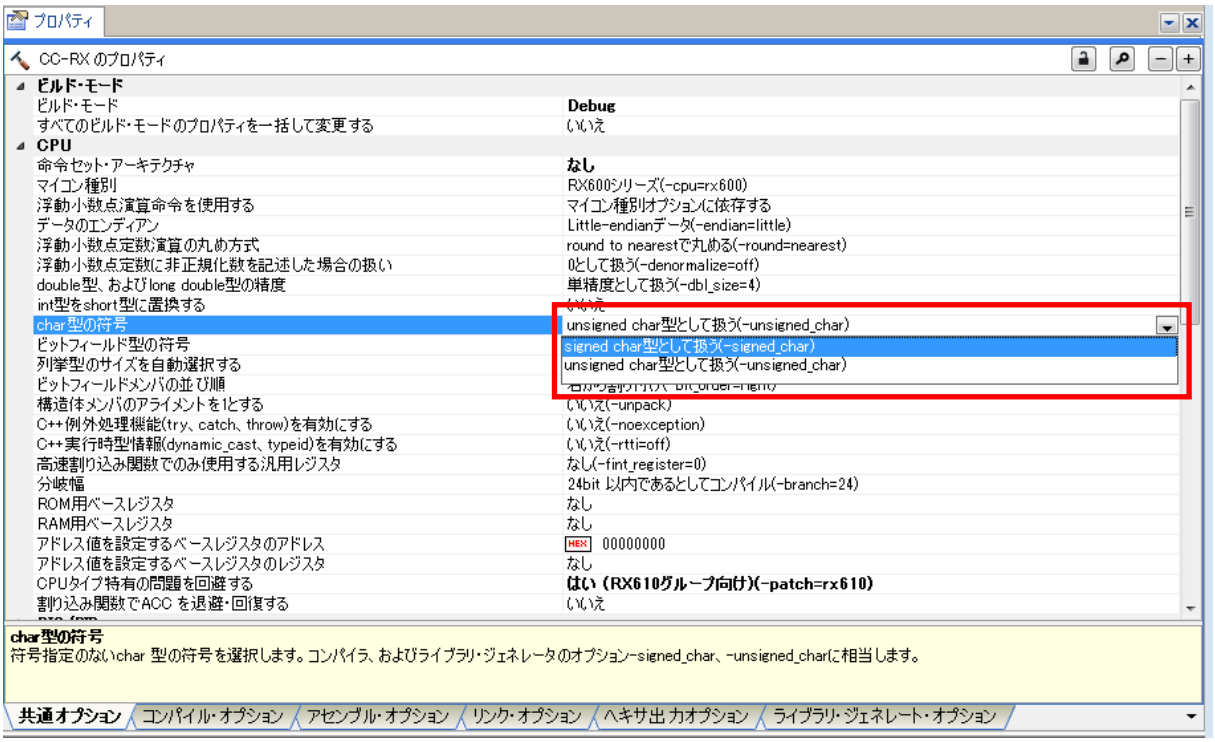

図 2-1

2.1.2　enum のサイズ指定

SuperH ファミリ用コンパイラで"auto_enum"オプションを指定して、enum 宣言した列挙型のデータを列挙値が収まる最小型としている場合、RX ファミリ用に移行する際には、RX ファミリ用コンパイラで"auto_enum"オプションを指定してください。

RX ファミリ用コンパイラでは"auto_enum"オプションを指定しない場合、列挙型サイズを signed long 型として扱います。

【書式】

auto_enum

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの“共通オプション”タブ内で次のように設定します。

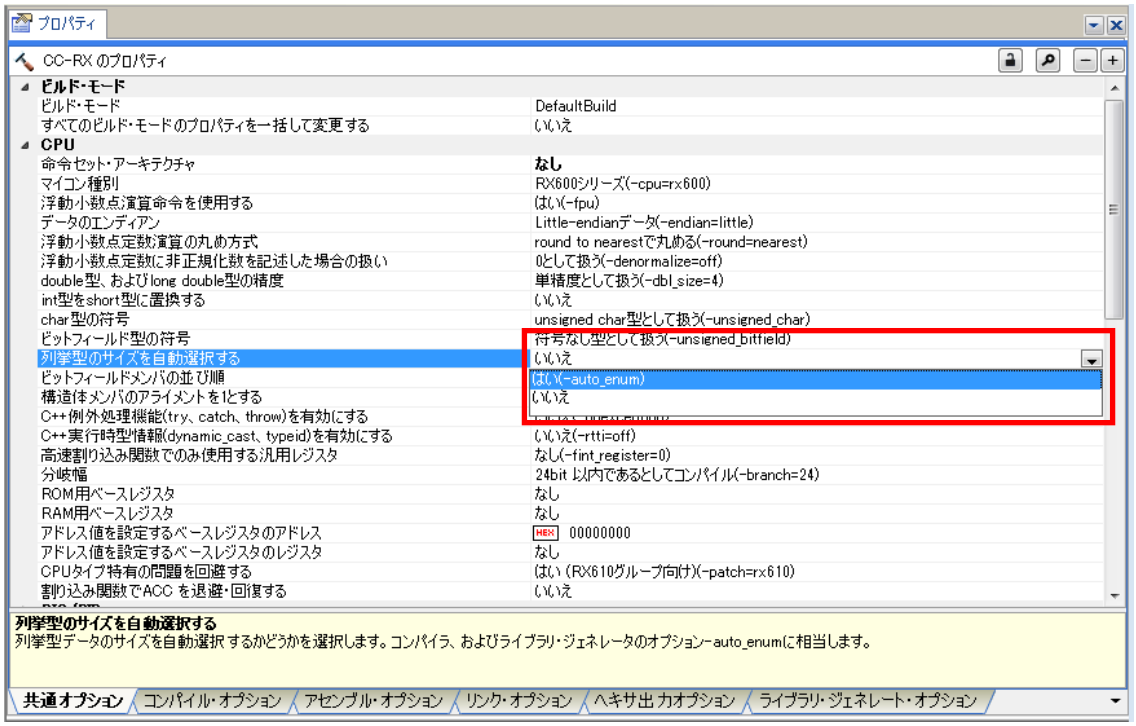

図 2-2

2.1.3　double 型のサイズ指定

SuperH ファミリ用コンパイラでは double 型のサイズは 8byte です。

対して RX ファミリ用コンパイラは、double 型のサイズはデフォルトでは 4byte です。

double 型のサイズが 8byte であることを前提に作成した SuperH ファミリ用のソースプログラムを RX ファミリ用に移行する際は、RX ファミリ用コンパイラで"dbl_size=8"オプションを指定してください。又、double 型のサイズが 4byte で作成した SuperH ファミリ用のソースプログラム(double=float を指定)を RX ファミリ用に移行する際には、RX ファミリ用コンパイラで"dbl_size=4"オプションを指定してください。

【書式】

dbl_size={ 4 | 8 }　　：デフォルトは 4

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの“共通オプション”タブ内で次のように設定します。

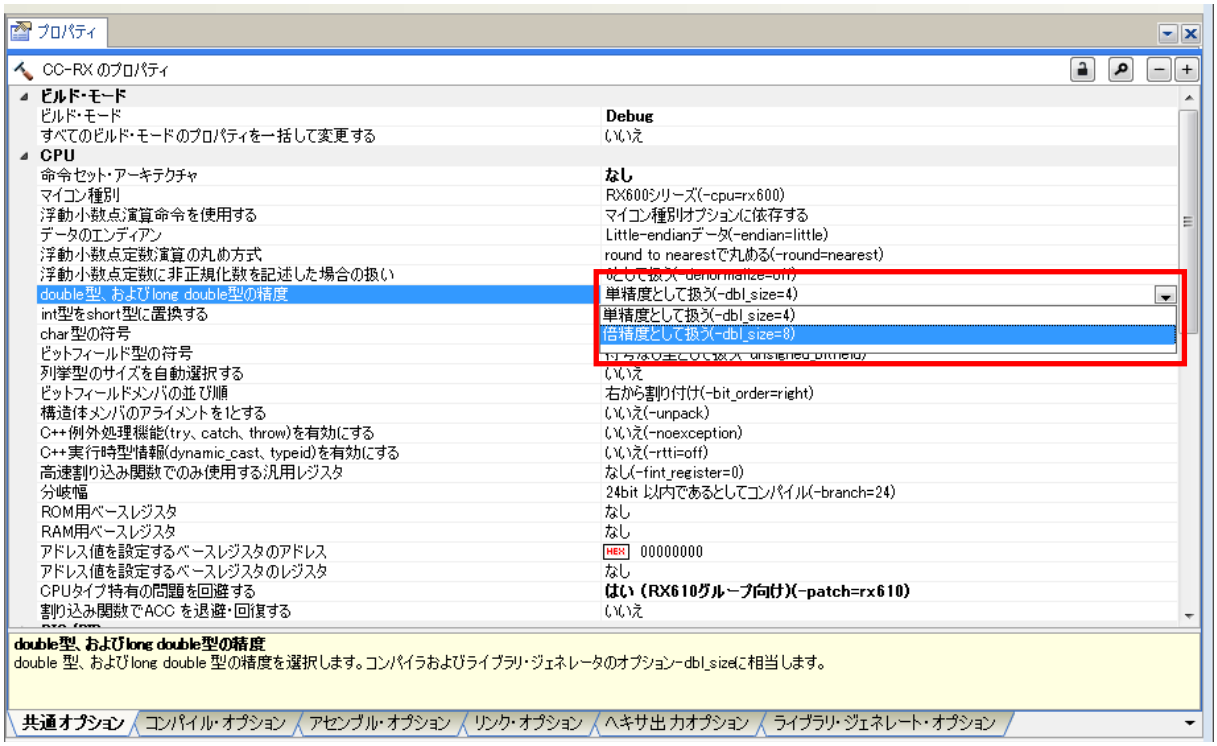

図 2-3

【備考】

SuperH ファミリ用コンパイラで double=float を指定したとき、long double 型のサイズが 8 バイトですが、RX ファミリ用コンパイラでは、dbl_size=4 を指定したとき、long double 型のサイズが 4 バイトになります。

2.1.4　エンディアン指定

SuperH ファミリ用コンパイラではデータのバイト並びが big endian になります。

対して RX ファミリ用コンパイラは、デフォルトでは little endian になります。

データのバイト並びが big endian であることを前提に作成した SuperH ファミリ用のソースプログラムを RX ファミリ用に移行する際は、RX ファミリ用コンパイラで"endian=big"オプションを指定してください。

【書式】

endian={ big | little }　　：デフォルトは little

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの"共通オプション"タブ内で次のように設定します。

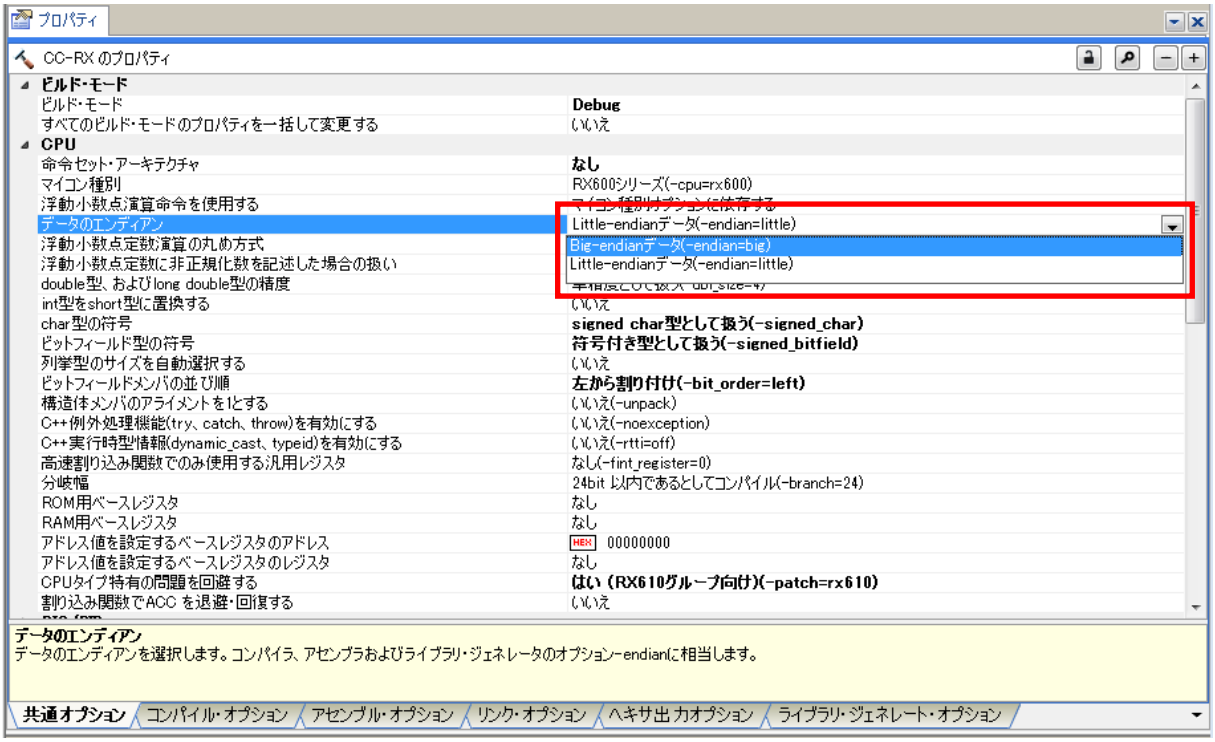

図 2-4

2.1.5　ビットフィールドメンバの符号指定

SuperH ファミリ用コンパイラでは符号指定のないビットフィールドメンバは、符号ありの型として扱います。

対して RX ファミリ用コンパイラでは、デフォルトでは符号なしの型として扱います。

符号指定のないビットフィールドメンバが、符号ありの型であることを前提に作成した SuperH ファミリ用のソースプログラムを RX ファミリ用に移行するには、RX ファミリ用コンパイラで"signed_bitfield"オプションを指定してください。

【書式】

signed_bitfield

unsigned_bitfield　　：デフォルトは unsigned_bitfield

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの"共通オプション"タブ内で次のように設定します。

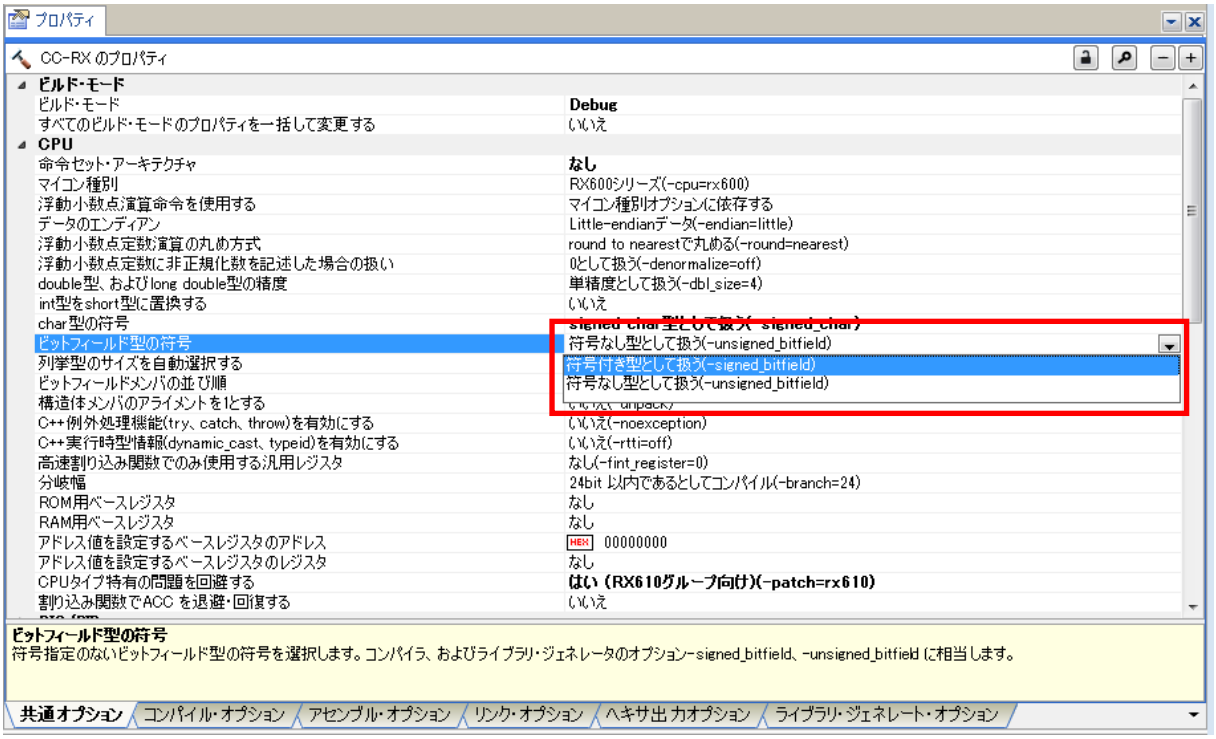

図 2-5

2.1.6　ビットフィールドメンバの割り付け順序指定

SuperH ファミリ用コンパイラではビットフィールドメンバを上位ビットから割り付けます。

対して RX ファミリ用コンパイラは、デフォルトでは下位ビットから割り付けます。

ビットフィールドメンバを上位から割り付けることを前提に作成した SuperH ファミリ用のソースプログラムを RX ファミリ用に移行するには、RX ファミリ用コンパイラで"bit_order=left"オプションを指定してください。

【書式】

bit_order={ left | right }　：デフォルトは right

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの"共通オプション"タブ内で次のように設定します。

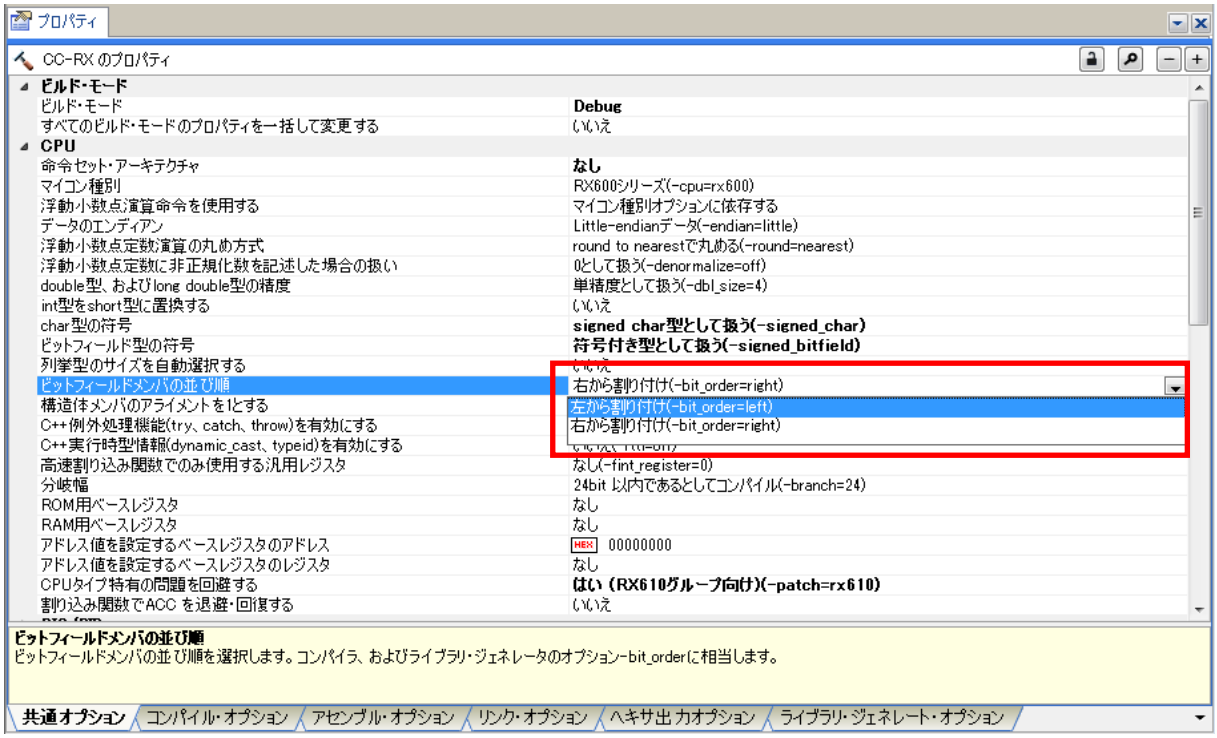

図 2-6

2.1.7　構造体の割り付け指定

SuperH ファミリ用コンパイラで"pack=1"オプションを指定して、構造体のアライメント数を 1 としている場合、RX ファミリ用に移行する際には RX ファミリ用コンパイラで"pack"オプションを指定してください。

【書式】

pack

unpack　　：デフォルトは unpack

[CS+でのオプション設定方法]

CC-RX（ビルド・ツール）プロパティの“共通オプション”タブ内で次のように設定します。

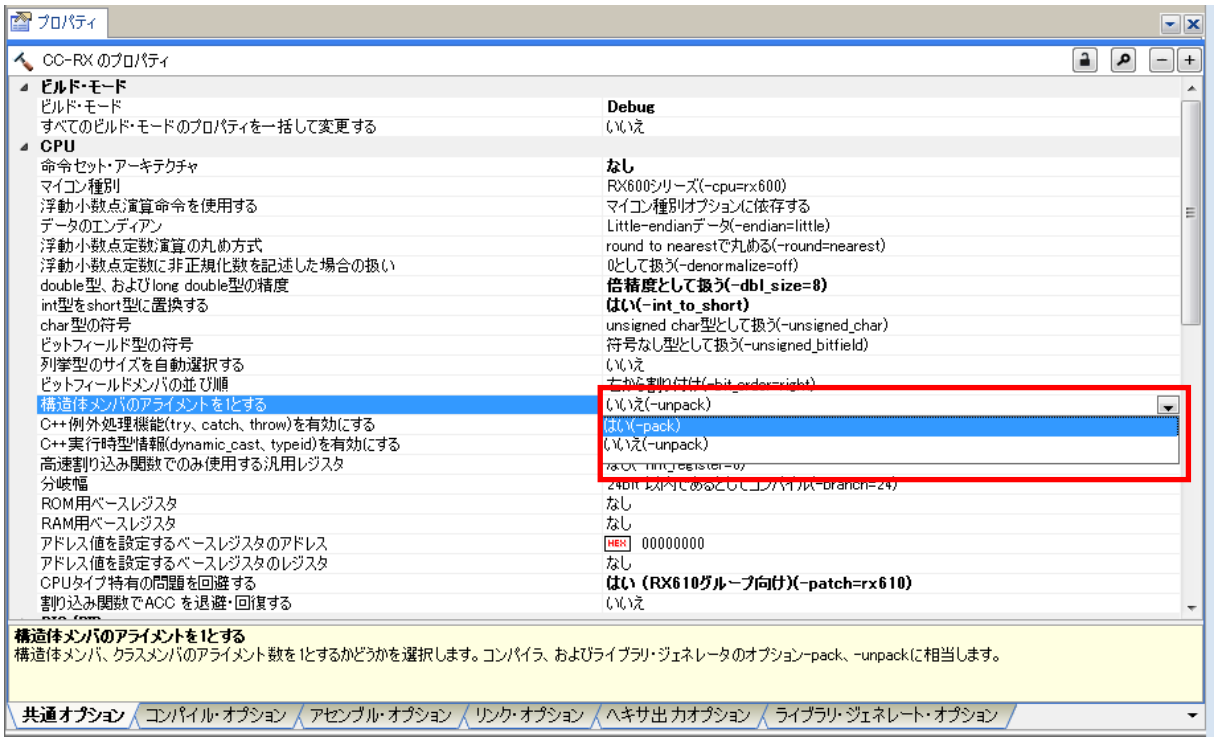

図 2-7

2.2　言語仕様

本章では、RX 移行時に変更が必要な言語仕様について説明します。

表　言語仕様一覧

No	機能	参照
1	char 型の符号	2.2.1
2	double 型のサイズ	2.2.2
3	エンディアン	2.2.3
4	ビットフィールドの割り付け順序	2.2.4
5	ビットフィールドの符号	2.2.5

2.2.1　char 型の符号

SuperH ファミリ用コンパイラでは符号指定のない char 型は、符号ありの signed char 型として扱います。対して RX ファミリ用コンパイラでは符号なしの unsigned char 型として扱います。

char 型が signed char 型であることを前提に作成した SuperH ファミリ用のソースプログラムを、RX ファミリ用に移行すると正しく動作しない場合があります。

【例】char 型の符号の有無で動作が異なる記述例

ソースコード

```
char a = -1;

void main(void)
{
  if (a < 0) {
    // char 型が符号ありで’a’を負数と解釈し、条件式は成立(SuperH)
  } else {
    // char 型が符号なしで’a’を正数と解釈し、条件式は不成立(RX)
  }
}
```

char 型が符号ありの signed char 型であることを前提に作成したソースプログラムを RX に移行するには、"signed_char"オプションを指定します。

オプション指定については「2.1.1 char 型の符号指定」を参照してください。

2.2.2　double 型のサイズ

SuperH ファミリ用コンパイラでは double 型のサイズは 8byte です。

対して RX ファミリ用コンパイラでは double 型のサイズは 4byte です。

double 型のサイズが 8byte であることを前提に作成した SuperH ファミリ用のソースプログラムを、RX ファミリ用に移行すると正しく動作しない場合があります。

【例】double 型のサイズの差異で動作が異なる記述例

ソースコード

```
double d1 = 1E30;
double d2 = 1E20;

void main(void)
{
  d1 = d1 * d1; // double 型のサイズが 4byte の場合、d1 * d1 がオーバフローする
  d2 = d2 * d2; // double 型のサイズが 4byte の場合、d2 * d2 がオーバフローする

  if (d1 > d2) {
    // double 型のサイズが 8byte の場合、正常な大小比較が行われる(SuperH)
  } else {
    //double 型のサイズが 4byte の場合、d1,d2 ともにオーバフローしているため大小比較が成立しない(RX)
  }
}
```

double 型のサイズが 8byte であることを前提に作成したソースプログラムを RX に移行するには、"dbl_size=8"オプションを指定します。

オプション指定については「2.1.3 double 型のサイズ指定」を参照してください。

2.2.3　エンディアン

SuperH ファミリ用コンパイラではデータのバイト並びが big endian になります。

対して RX ファミリ用コンパイラでは little endian になります。

データのバイト並びが big endian であることを前提に作成した SuperH ファミリ用のソースプログラムを、RX ファミリ用に移行すると正しく動作しない場合があります。

【例】エンディアンの差異で動作が異なる記述例

ソースコード

```
typedef union{
  short data1;
  struct {
    unsigned char upper;
    unsigned char lower;
  } data2;
} UN;

UN u = { 0x7f6f };

void main(void)
{
  if (u.data2.upper == 0x7f && u.data2.lower == 0x6f) {
    // データバイトの並びが big endian の場合(SuperH)
  } else {
    // データバイトの並びが little endian の場合(RX)
  }
}
```

データのバイト並びが big endian であることを前提に作成したソースプログラムを RX に移行するには、"endian=big"オプションを指定します。

オプション指定については「2.1.4 エンディアン指定」を参照してください。

2.2.4　ビットフィールドの割り付け順序

SuperH ファミリ用コンパイラではビットフィールドメンバを上位ビットから割り付けます。

対して RX ファミリ用コンパイラでは下位ビットから割り付けます。

ビットフィールドメンバを上位ビットから割り付けることを前提に作成した SuperH ファミリ用のソースプログラムを、RX ファミリ用に移行すると正しく動作しない場合があります。

【例】ビットフィールドの割付順序の差異で動作が異なる記述例

ソースコード

```
union {
  unsigned char c1;
  struct {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
  } b;
} un;

void bit_order(void)
{
  un.c1 = 0xc0;
  if ((un.b.b0 == 1) && (un.b.b1 == 1) &&
    (un.b.b2 == 0) && (un.b.b3 == 0)) {
    // ビットフィールドメンバを上位から割り付けるの場合(SuperH)
  } else {
    // ビットフィールドメンバを下位から割り付けるの場合(RX)
  }
}
```

SuperH の割付(left)

1	1	0	0	0	0	0	0
b0	b1	b2	b3				

上位ビットに割り付き、設定値が b0,b1 で参照可能

RX の割付(right)

1	1	0	0	0	0	0	0
				b3	b2	b1	b0

下位ビットに割り付き、設定値が参照不可能

ビットフィールドメンバを上位ビットから割り付けることを前提に作成したソースプログラムを RX に移行するには、"bit_order=left"オプションを指定します。

オプション指定については「2.1.6 ビットフィールドメンバの割り付け順序指定」を参照してください。

2.2.5　ビットフィールドの符号

SuperH ファミリ用コンパイラでは符号指定のないビットフィールドメンバは、符号ありの型として扱います。

対して RX ファミリ用コンパイラでは符号なしの型として扱います。

符号指定のないビットフィールドメンバが符号ありの型であることを前提に作成した SuperH ファミリ用のソースプログラムを、RX ファミリ用に移行すると正しく動作しない場合があります。

【例】ビットフィールドメンバの符号の有無で動作が異なる記述例

ソースコード

```
struct S {
  int a : 15;
} s = { -1 };

void main(void)
{
  if (s.a < 0) {
    // ビットフィールドメンバが符号ありで’s.a’を負数と解釈し、条件式は成立(SuperH)
  } else {
    // ビットフィールドメンバが符号なしで’s.a’を正数と解釈し、条件式は不成立(RX)
  }
}
```

符号指定のないビットフィールドメンバが符号ありの型であることを前提に作成したソースプログラムを RX に移行するには、"signed_bitfield"オプションを指定します。

オプション指定については「2.1.5 ビットフィールドメンバの符号指定」を参照してください。

2.2.6　拡張言語仕様

(1) #pragma pack の対応

SuperH ファミリ用コンパイラで #pragma pack を使用している場合、RX ファミリ用コンパイラでは指定を変更する必要があります。

表　#pragma pack 対応一覧

SuperH	RX	備考
#pragma pack1	#pragma pack	アライメント数を 1 とする
#pragma pack4	#pragma unpack	
#pragma unpack	#pragma packoption	pack オプションに従う

(2) evenaccess の対応

SuperH ファミリ用コンパイラでは、volatile 宣言された変数は、その型のサイズでアクセスすることを保障します。

対して RX ファミリ用コンパイラでは、変数の型のサイズでアクセスすることを保障するためには、evenaccess を以下の書式に従って使用する必要があります。

__evenaccess <型指定子> <変数名>

<型指定子> __evenaccess <変数名>

2.2.7　プリデファインドマクロ

SuperH ファミリ用コンパイラと RX ファミリ用コンパイラでは、オプション指定時に定義されるプリデファインドマクロが異なりますので注意してください。

オプションを対応させた場合には、以下の表に従って、RX ファミリ用コンパイラのプリデファインドマクロ名に変更が必要です。

表　SuperH のプリデファインドマクロ

オプション	プリデファインドマクロ
endian=big	_BIG
endian=little	_LIT
double=float	_FLT __FLT__
denormalize=on	_DON
round=nearest	_RON

表　RX のプリデファインドマクロ

オプション	プリデファインドマクロ
endian=big	__BIG
endian=little	__LIT
double=float	__DBL4
denormalize=on	__DON
round=nearest	__RON

3. 移行サンプルプロジェクト

シミュレータデバッガで動作確認ができる SuperH サンプルプロジェクトを、RX へ移行する手順を説明します。

3.1 SuperH サンプルプロジェクト概要

SuperH サンプルプロジェクト'SH_Sample'は大きく分けて初期化などを行う前後処理と、主要な処理を行うメイン処理に分かれています。

メイン処理を構成するファイルを以下に示します。

表　メイン処理ファイル一覧

No	機能	ファイル名	参照
1	char 型の符号	SH_sign_char.c	3.2.5 (1)
2	ビットフィールドメンバの符号	SH_sign_bit_field.c	3.2.5 (2)
3	ビットフィールドメンバの割付	SH_bit_order.c	3.2.5 (3)
4	エンディアン	SH_endian.c	3.2.5 (4)
5	double 型のサイズ	SH_double_size.c	3.2.5 (5)
6	main 関数	SH_Sample.c	-

3.2 SuperH サンプルプロジェクト RX 移行手順

3.2.1 RX プロジェクトの作成

SuperH サンプルプロジェクトの移行先となる RX の新規プロジェクトを作成します。

(1) サンプルプロジェクトを読み込み

[サンプルプロジェクトを読み込む]の” RX”タブで”RX610_Tutorial_DebugConsole"を選択します。

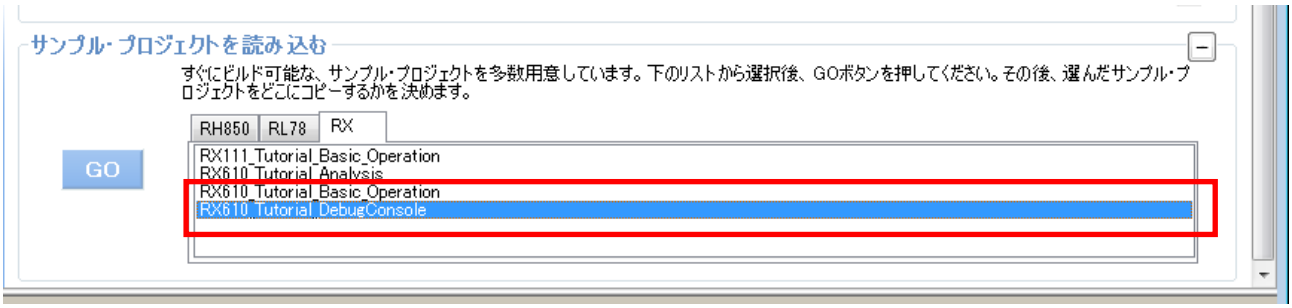

図 3-1

(2) サンプルプロジェクトをコピーする場所を選択

サンプルプロジェクトをコピーするフォルダーを選択します。

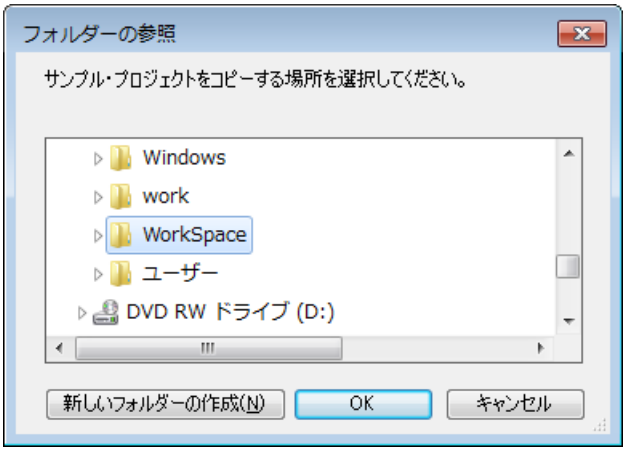

図 3-2

(3) 使用するデバッグ・ツールの選択

使用するデバッグ・ツールに“RX シミュレータ”を選択します。

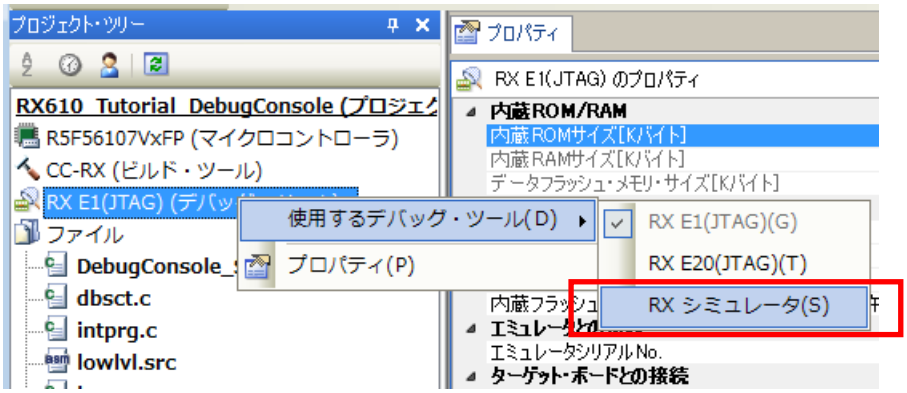

図 3-3

(4) ストリーム入出力モードの選択

デバッグ・ツールプロパティの”デバッグ・ツール設定”タブの[ストリーム入出力]カテゴリ内で次のように設定します。

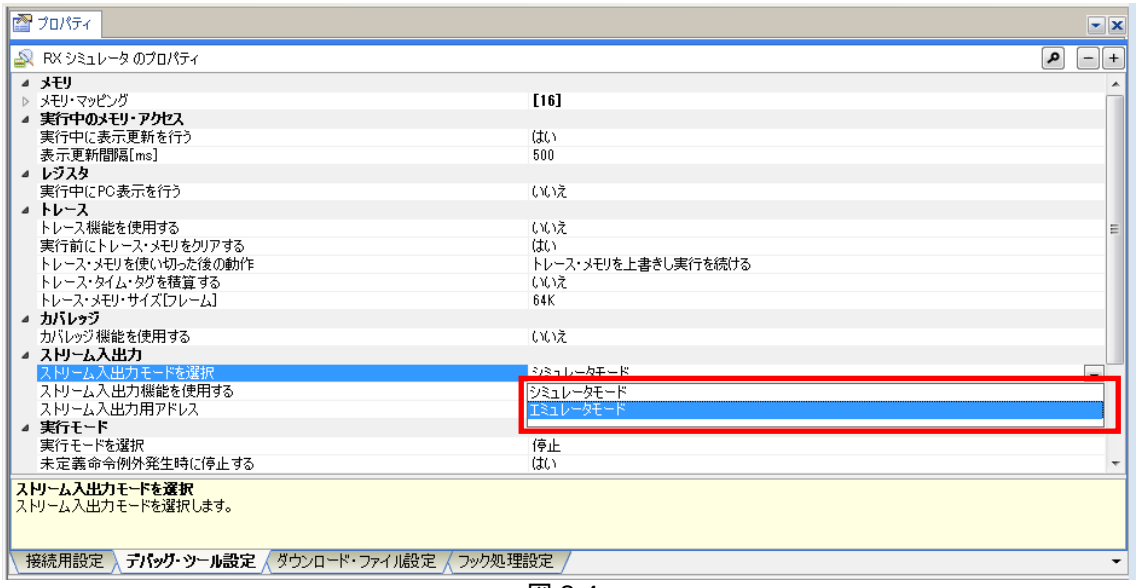

図 3-4

3.2.2　メイン処理ソースファイル移行

「3.1 SuperH サンプルプロジェクト概要」で説明した SuperH サンプルプロジェクトのメイン処理を構成するファイルを、作成した RX プロジェクトにコピー・登録します。

(1) SuperH サンプルプロジェクトフォルダからファイルをコピー

「3.1 SuperH サンプルプロジェクト概要」で説明した 6 つのファイルをコピーします。

[コピー前]　　[コピー後]

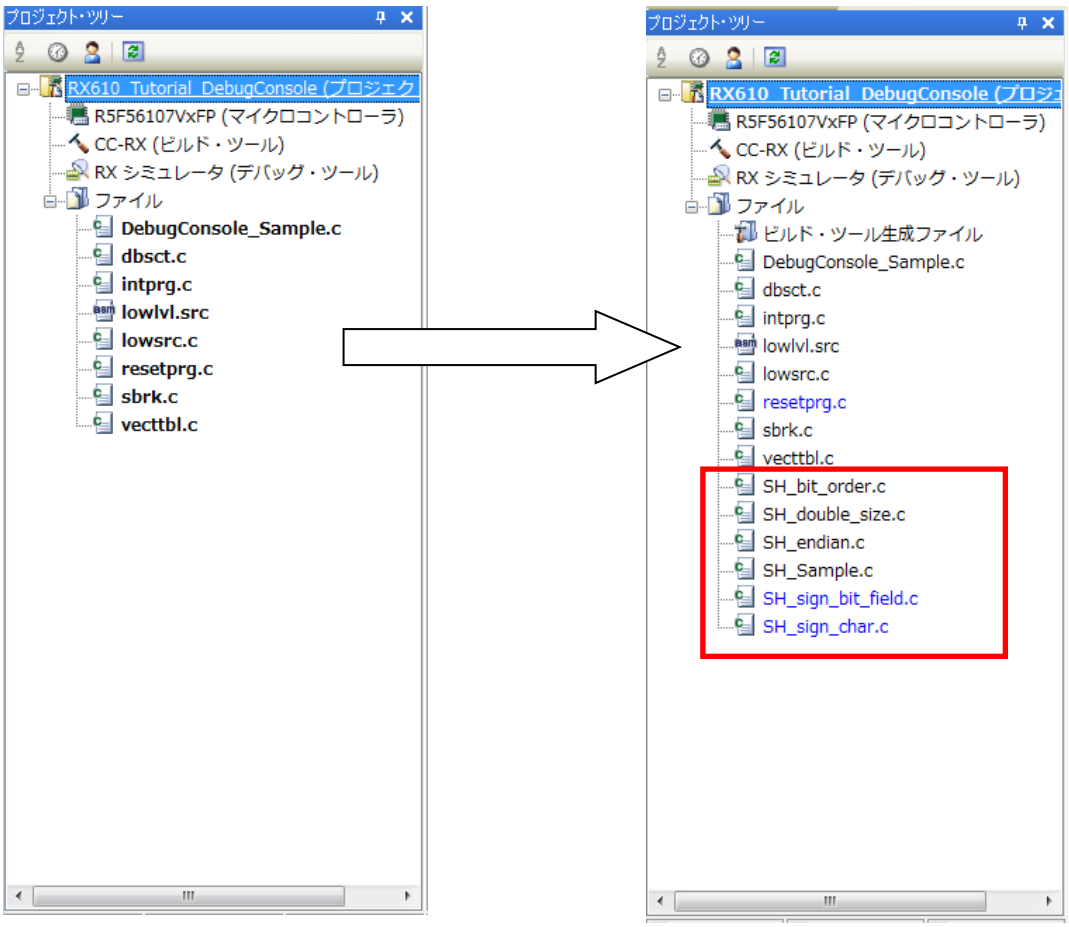

図 3-5

(2) コピーしたファイルをプロジェクトに登録する

CS+のメニュー[プロジェクト → 追加 → 既存のファイルを追加]を選択すると表示されるダイアログで次のように選択します。

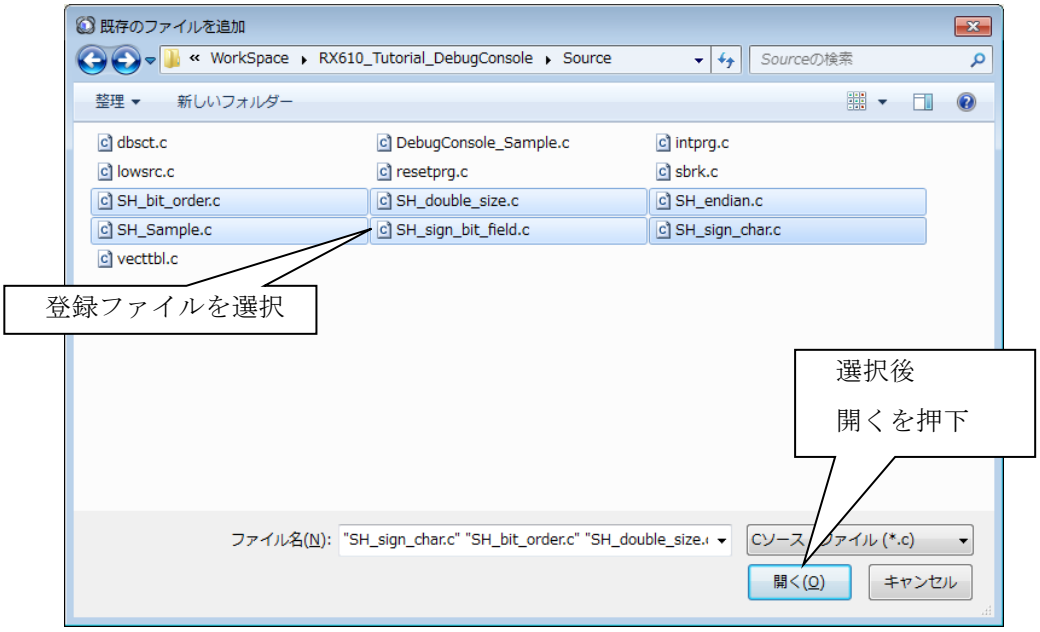

図 3-6

(3) 不要ファイルの登録削除

RX のサンプルプロジェクトにある main 関数ファイル” DebugConsole_Sample.c”は不要なので、プロジェクトから削除します。(main 関数ファイルは SuperH プロジェクトからコピーするため)

プロジェクト・ツリーから” DebugConsole_Sample.c”を選択して、[プロジェクトから外す]を選択します。

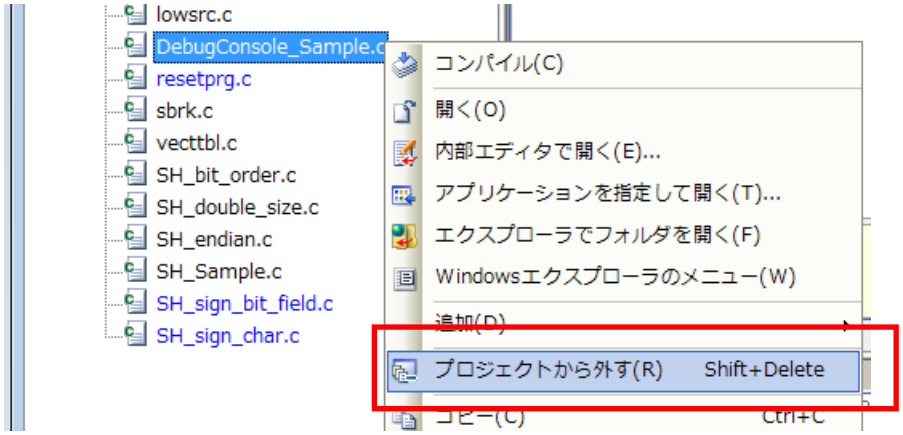

図 3-7

3.2.3　ビルド

メイン処理ファイルをコピー・登録した RX プロジェクトをビルドします。

CS+のメニュー[ビルド → ビルド・プロジェクト]を選択するとビルドを開始します。

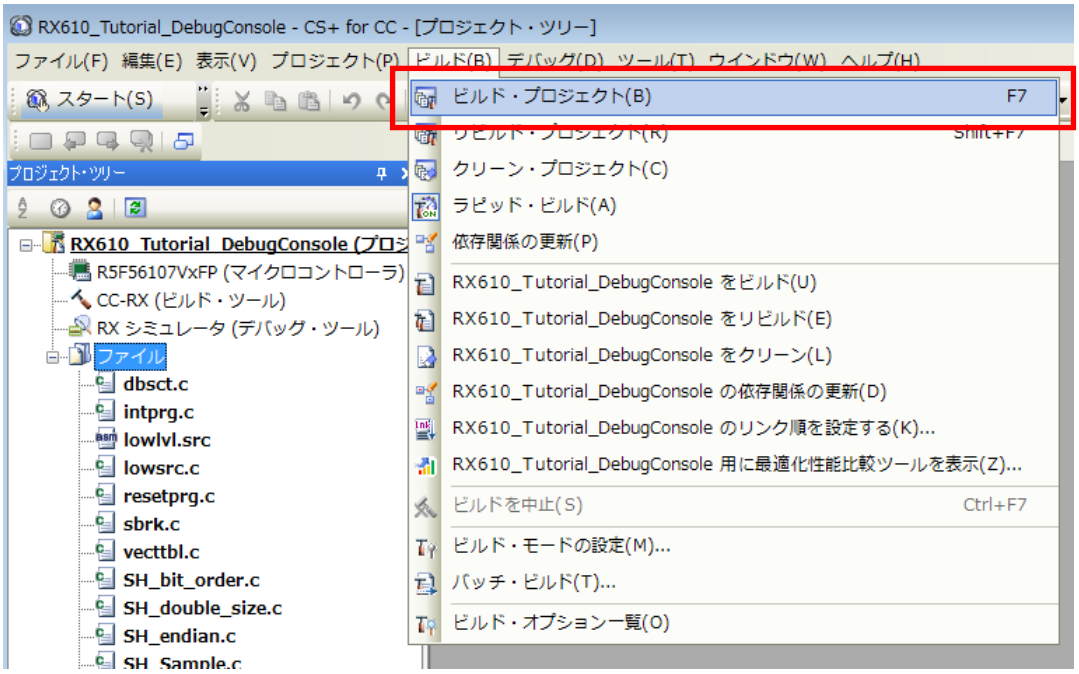

図 3-8

3.2.4　シミュレータ実行

ビルドした RX プロジェクトのロードモジュールをシミュレータで実行します。

(1) デバッグ・コンソール設定

ソースプログラムは実行結果を標準出力に表示します。

標準出力を表示するには'デバッグ・コンソール'プラグインを有効にする必要があります。

CS+のメニュー[ツール → プラグインの管理]で表示されるダイアログから以下のように選択してください。

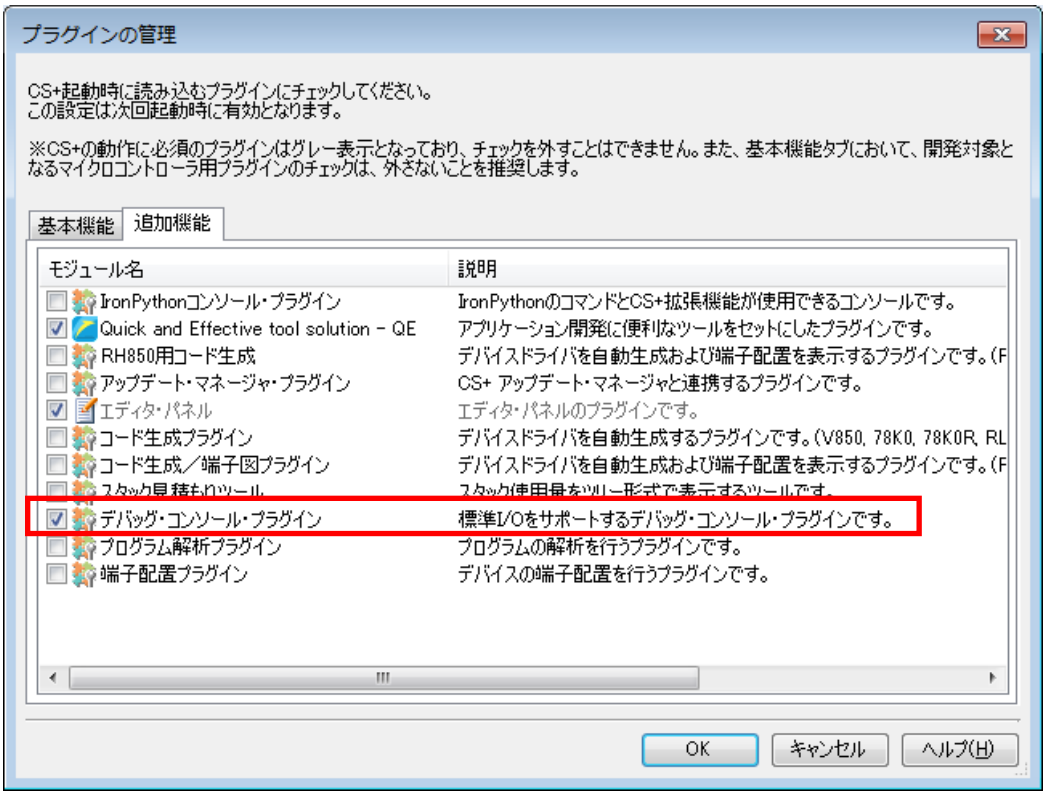

図 3-9

(2) デバッグ・ツールへのダウンロード

CS+のメニュー[デバッグ → デバッグ・ツールへのダウンロード]を選択するとビルドしたロードモジュールをデバッグ・ツールへダウンロードします。

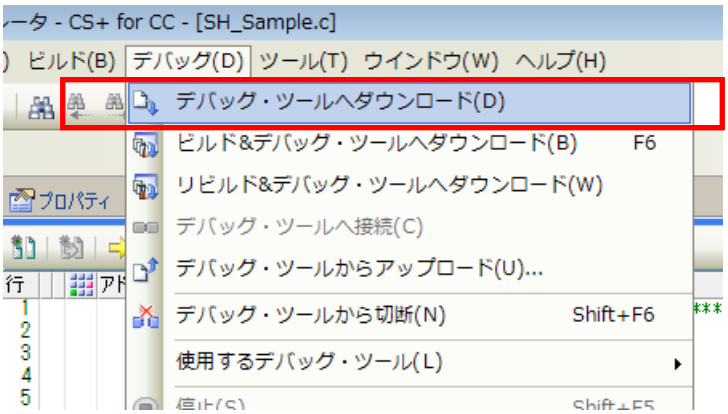

図 3-10

(3) 'デバッグ・コンソール'パネルの表示

標準出力を表示するために'デバッグ・コンソール'パネルを有効にする必要があります。

CS+のメニュー[表示 → デバッグ・コンソール]を選択すると'デバッグ・コンソール'パネルが表示されます。

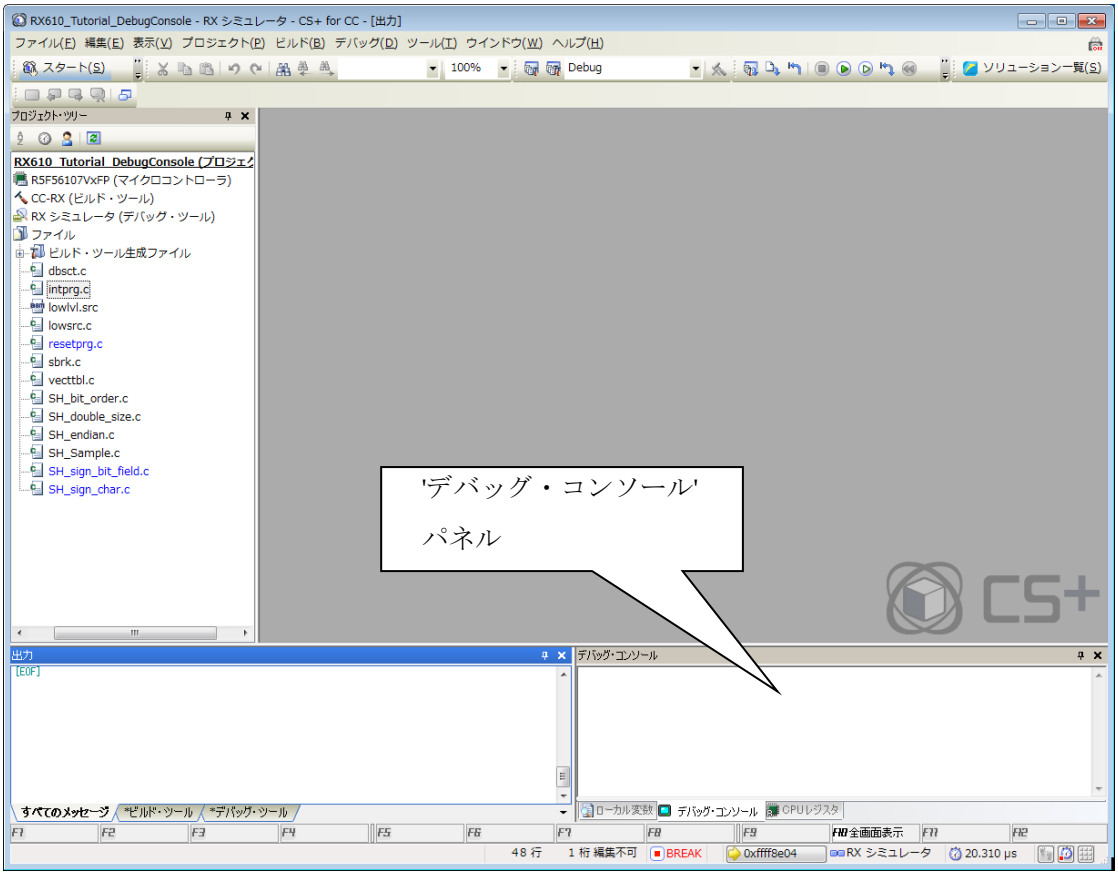

図 3-11

(4) シミュレータ実行

CS+のメニュー[デバッグ → 実行]を選択するとソースプログラムがシミュレータにより実行され、'デバッグ・コンソール'パネルにソースプログラムの標準出力が表示されます。

表示結果を見ると "NG"があり、結果が不正であることがわかります。

＜デバッグ・コンソールパネル＞

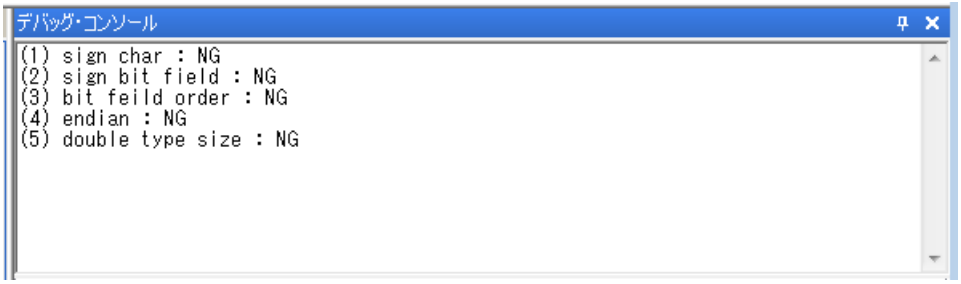

図 3-12

表　デバッグ・コンソール出力結果

項目	OK	NG
(1) 符号指定のない char 型	signed	unsigned
(2) 符号指定のないビットフィールドメンバ	signed	unsigned
(3) ビットフィールドメンバ割り付け順序	上位ビットから	下位ビットから
(4) endian	big	little
(5) double 型のサイズ	8byte	4byte

3.2.5　オプション設定

シミュレータ実行の結果は、SuperH ファミリ用と RX ファミリ用コンパイラの処理系定義の仕様の違いにより、“NG”となっています。

本章では、SuperH ファミリ用から移行した RX ファミリ用プロジェクトをサンプルとして、処理系定義の仕様の違いを、オプション指定を変更して解決する方法を示します。

(1) char の符号

サンプルソースプログラム"SH_sign_char.c"の実行結果が"NG"となった場合、"unsigned_char"オプション指定が互換性に問題があることを示します。

SuperH ファミリ用コンパイラでは符号指定のない char 型は、符号ありの signed char 型として扱います。

対して RX ファミリ用コンパイラでは符号なしの unsigned char 型として扱います。

サンプルソースプログラム"SH_sign_char.c"は符号指定のない char 型が signed char 型であることを前提として記述してあるため、"unsigned_char"オプションが指定されている場合、SuperH とは異なった動作結果となります。

【サンプルソースプログラム"SH_sign_char.c"】

ソースコード

```
struct S {
  char a;
} s = { -1 };

void sign_char(void)
{
  printf("(1) sign char : ");

  if (s.a < 0) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

char 型が符号ありの signed char 型であることを前提に作成したソースプログラムを RX に移行するには、"signed_char"オプションを指定します。

オプション指定の詳細については「2.1.1 char 型の符号指定」を参照してください。

また、作成した RX プロジェクトのオプション指定を変更してください。

(2) ビットフィールドの符号

サンプルソースプログラム"SH_sign_bit_field.c"の実行結果が"NG"となった場合、" unsigned_bitfield"オプション指定が互換性に問題があることを指摘しています。

SuperH ファミリ用コンパイラでは符号指定のないビットフィールドメンバは、符号ありの型として扱います。

対して RX ファミリ用コンパイラでは符号なしの型として扱います。

サンプルソースプログラムの"SH_sign_bit_field.c"は、符号指定のないビットフィールドメンバが符号ありの型であることを前提として記述してあるため、" unsigned_bitfield"オプションが指定されている場合、SuperH とは異なった動作結果となります。

【サンプルソースプログラム"SH_sign_bit_field.c"】

ソースコード

```
struct S {
  int a : 15;
} bit = { -1 };

void sign_bit_field(void)
{
  printf("(2) sign bit field : ");
  if (bit.a < 0) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

符号指定のないビットフィールドメンバが、符号ありであることを前提に作成したソースプログラムを RX に移行するには、"signed_bitfield"オプションを指定します。

オプション指定の詳細については「2.1.5 ビットフィールドメンバの符号指定」を参照してください。

また、作成した RX プロジェクトのオプション指定を変更してください。

(3) ビットフィールドの割付順序

サンプルソースプログラム"SH_bit_order.c"の実行結果が"NG"となった場合、"bit_order=right"オプション指定が互換性に問題があることを指摘しています。

SuperH ファミリ用コンパイラではビットフィールドメンバを上位ビットから割り付けます。

対して RX ファミリ用コンパイラでは下位ビットから割り付けます。

サンプルソースプログラムの"SH_bit_order.c"は、ビットフィールドメンバを上位ビットから割り付けることを前提として記述してあるため、"bit_order=right"オプションが指定されている場合、SuperH とは異なった動作結果となります。

【サンプルソースプログラム"SH_bit_order.c"】

ソースコード

```
union {
  unsigned char c1;
  struct {
    unsigned char b0 : 1;
    unsigned char b1 : 1;
    unsigned char b2 : 1;
    unsigned char b3 : 1;
  } b;
} un;

void bit_order(void)
{
  printf("(3) bit field order : ");

  un.c1 = 0xc0;
  if ((un.b.b0 == 1) && (un.b.b1 == 1) &&
   (un.b.b2 == 0) && (un.b.b3 == 0)) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

ビットフィールドメンバを上位から割り付けることを前提に作成したソースプログラムを RX に移行するには、" bit_order=left"オプションを指定します。

オプション指定の詳細については「2.1.6 ビットフィールドメンバの割り付け順序指定」を参照してください。

また、作成した RX プロジェクトのオプション指定を変更してください。

(4) エンディアン

サンプルソースプログラム"SH_endian.c"の実行結果が"NG"となった場合、"endian=little"オプション指定が互換性に問題があることを指摘しています。

SuperH ファミリ用コンパイラではデータのバイト並びが big endian になります。

対して RX ファミリ用コンパイラでは little endian になります。

サンプルソースプログラムの"SH_endian.c"は、データのバイト並びが big endian であることを前提として記述してあるため、"endian=little"オプションが指定されている場合、SuperH とは異なった動作結果となります。

【サンプルソースプログラム"SH_endian.c"】

ソースコード

```
typedef union{
  short data1;
  struct {
    unsigned char upper;
    unsigned char lower;
  } data2;
} UN;

UN u = { 0x7f6f };

void endian(void)
{
  printf("(4) endian : ");

  if (u.data2.upper == 0x7f && u.data2.lower == 0x6f) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

データのバイト並びが big endian であることを前提に作成したソースプログラムを RX に移行するには、"endian=little"オプションを指定します。

オプション指定の詳細については「2.1.4 エンディアン指定」を参照してください。

また、作成した RX プロジェクトのオプション指定を変更してください。

(5) double 型のサイズ

サンプルソースプログラム"SH_double_size.c"の実行結果が"NG"となった場合、"dbl_size=4"オプション指定が互換性に問題があることを指摘しています。

SuperH ファミリ用コンパイラでは double 型のサイズは 8byte です。

対して RX ファミリ用コンパイラでは double 型のサイズは 4byte です。

サンプルソースプログラムの"SH_double_size.c"は、double 型のサイズが 8byte であることを前提として記述してあるため、"dbl_size=4"オプションが指定されている場合、SuperH とは異なった動作結果となります。

【サンプルソースプログラム"SH_double_size.c"】

ソースコード

```
double d1 = 1E30;
double d2 = 1E20;

void double_size(void)
{
  d1 = d1 * d1;
  d2 = d2 * d2;

  printf("(5) double type size : ");

  if (d1 > d2) {
    printf("OK¥n");
  } else {
    printf("NG¥n");
  }
}
```

double 型のサイズが 8byte であることを前提に作成したソースプログラムを RX に移行するには、"dbl_size=8"オプションを指定します。

オプション指定の詳細については「2.1.3 double 型のサイズ指定」を参照してください。

また、作成した RX プロジェクトのオプション指定を変更してください。

3.2.6　再ビルド

(1) シミュレータのエンディアン設定

endian オプションで endian を little から big へ変更しているので、シミュレータについても endian を big に変更する必要があります。

デバッグ・ツールとの接続中は、シミュレータの endian 変更はできません。まずは CS+のメニュー[デバッグ → デバッグ・ツールから切断]を選択してください。

デバッグ・ツールプロパティの"接続用設定"タブの[エンディアン]カテゴリ内で次のように設定します。

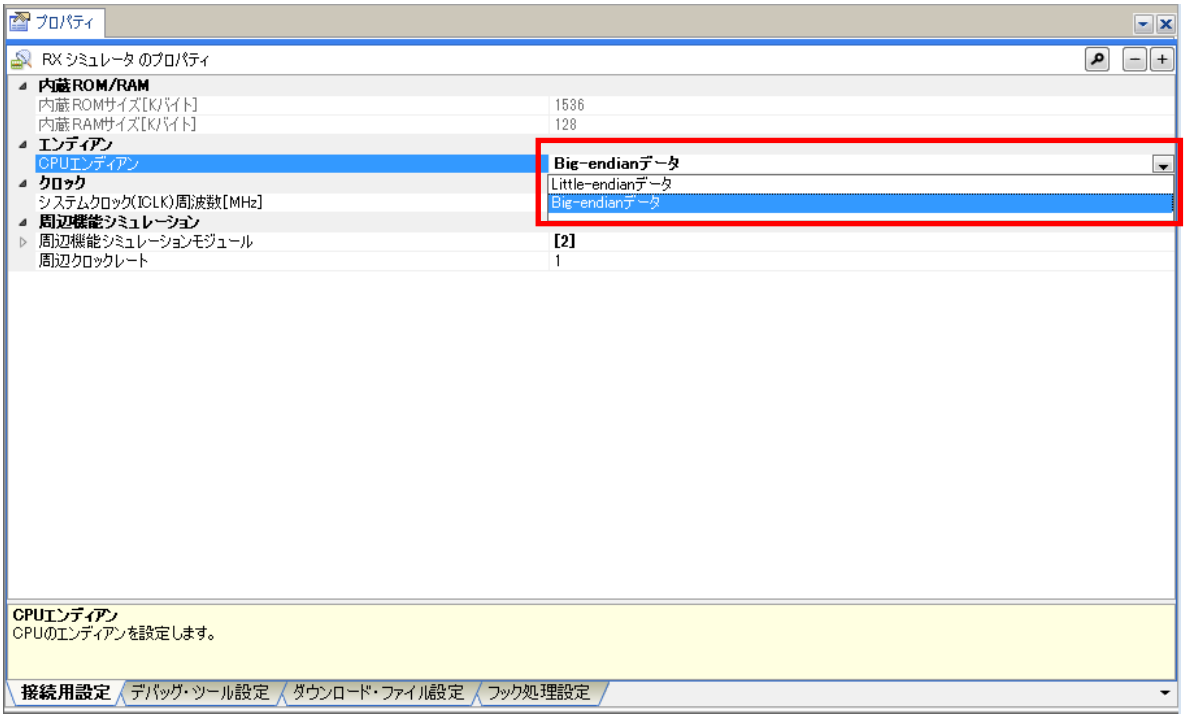

図 3-13

3.2.7　実行結果確認

再ビルドしたロードモジュールをシミュレータで実行し、実行結果が正しくなっていることを確認します。

シミュレータ実行方法については' 3.2.4 (4)　シミュレータ実行'を参照してください。

シミュレータにより実行され、'デバッグ・コンソール'パネルにソースプログラムの標準出力が表示されます。

表示結果が"OK"であることを確認してください。"NG"がある場合、オプション指定を再確認してください。

＜デバッグ・コンソールパネル＞

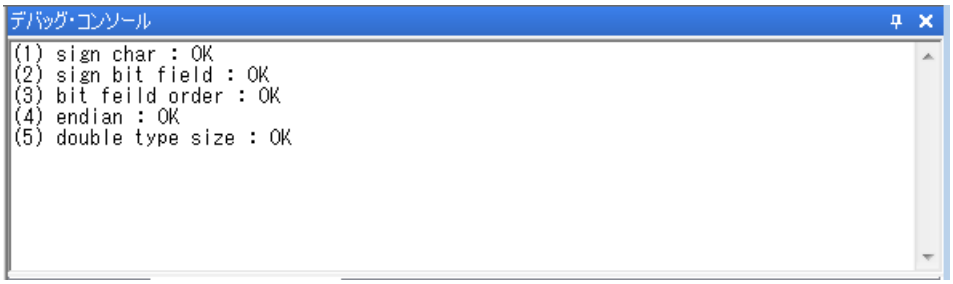

図 3-14

4. 対応一覧

4.1 オプション

SuperH ファミリ用 C/C++コンパイラの機種依存となるオプションは、RX ファミリ用 C/C++コンパイラと互換性がありません。

以下の表に、オプション対応一覧を示します。英大文字は、短縮形指定時の文字を示します。RX には短縮形はありません。

RX と形式が異なるオプションについては、指定方法を変更する必要があります。また、互換性がないオプションについては、指定を除去してください。

表　オプション対応一覧

SuperH	RX	備考
Include = <パス名>[,…]	include = <パス名>[,…]	
PREInclude = <ファイル名>[,…]	preinclude = <ファイル名>[,…]	
DEFine = <sub>[,…]	define = <sub>[,…]	
MEssage | NOMEssage	message | nomessage	
FILE_INLINE_PATH= <パス名>[,...]	file_inline_path=<パス名>[,…]	
CHAnge_message =<sub>[,…]	change_message=<sub>[,…]	
PREProcessor[= <ファイル名>]	output = prep	
Code ={ Machinecode | Asmcode }	output= { obj | src }	
DEBug	debug	
SEction = <sub>[,…]	section = <sub>[,…]	
STring = { Const | Data }	—	
OBjectfile = <ファイル名>	output = obj = <ファイル名>	
Template = { None |Static |Used |ALl |AUto }	—	
ABs16 =<sub>[,…]	—	SuperH の ABS20, ABS28, ABS32 も同様
DIvision = Cpu ={ Inline | Runtime }	—	
IFUnc	—	
ALIGN16	—	SuperH の ALIGN32 も同様
TBR [=<セクション名>]	—	
BSs_order = { DEClaration | DEFinition }	—	
STUff [={Bss | Data | Const} [,…]]	nostuff[= { B | D | C } [,…]]	SuperH で stuff 指定しないものを RX で nostuff 指定する
Listfile [= <ファイル名>]	listfile[=<ファイル名>]	
SHow = <sub>[,…]	show = <sub>[,…]	<sub>オプションの指定方法は異なる
OPtimize = 0	optimize = 1	最適化なし
OPtimize = 1	optimize = 2	最適化あり
OPtimize = Debug_only	optimize = 0	デバッグ情報の精度を優先したコード
SPeed	speed	
SIze	size	
NOSPeed	—	
Goptimize	goptimize	

MAP = <ファイル名>	map=<ファイル名>	
SMap	smap	
GBr = { Auto | User }	—	
CAse = { Ifthen | Table }	case = { ifthen | table | auto }	
SHIft = { Inline | Runtime }	—	
BLOckcopy = { Inline | Runtime }	—	
Unaligned = { Inline | Runtime }	—	
INLine[= <数値>]	inline[= <整数>]	
FILe_inline= <ファイル名>[,…]	file_inline = <ファイル名>[, …]	
GLOBAL_Volatile={ 0 | 1 }	novolatile | volatile	
OPT_Range={All | NOLoop | NOBlock }	—	
DEL_vacant_loop={ 0 | 1 }	—	
MAX_unroll=<数値>	—	
INFinite_loop={ 0 | 1 }	—	
GLOBAL_Alloc={ 0 | 1 }	—	
STRUCT_Alloc={ 0 | 1 }	—	
CONST_Var_propagate={ 0 | 1 }	const_copy	
CONST_Load={ Inline | Literal }	—	
SChedule={ 0 | 1 }	schedule | noschedule	
SOftpipe	—	
SCOpe	scope	
NOSCOpe	noscope	
LOGIc_gbr	—	
ECpp	lang = ecpp	
DSpc	—	
COMment = { Nest | NONest }	comment = { nest | nonest }	
Macsave = { 0 | 1 }	—	
SAve_cont_reg={ 0 | 1 }	—	
RTnext	—	
LOop	loop[=<数値>]	
APproxdiv	approxdiv	
PAtch=7055	—	
FPScr = { Safe | Aggressive }	—	
Volatile_loop	—	
AUto_enum	auto_enum	
ENAble_register	enable_register	
STRIct_ansi	—	
FDIv	—	
FIXED_Const	—	
FIXED_Max	—	
FIXED_Noround	—	
REPeat	—	
SIMple_float_conv	simple_float_conv	
CPu=<CPU 種別>	cpu=<CPU 種別>	<CPU 種別>は異なる
ENdian = { Big | Little }	endian = { big | little }	
FPu = { Single | Double }	—	
Round = { Zero | Nearest }	round = { zero | nearest }	
DENormalize = { OFF | ON }	denormalize = { off | on }	

Pic = { 0 | 1 }	—	
DOuble = Float	dbl_size = 4	
BIt_order={ Left | Right }	bit_order = { left | right }	
PACK={ 1 | 4 }	pack | unpack	
EXception	exception	
RTTI = { ON | OFF }	rtti= { on | off}	
DIvision = { Cpu | Peripheral | Nomask }	—	
LAng = { C | CPp }	lang = { c | cpp | ecpp | c99 }	
LOGO | NOLOGO	logo | nologo	
Euc | SJis | LATin1	euc | sjis | latin1 | utf8	
OUtcode = { EUc | SJis }	outcode = { euc | sjis | utf8 }	
SUbcommand = <ファイル名>	subcommand = <ファイル名>	
STUFF_GBR	—	
ALIGN4={ALL|LOOP|INMOSTLOOP}	—	
CPP_NOINLINE	—	
CONST_VOLATILE={DATA|CONST}	—	

4.2　#pragma

以下の SuperH ファミリ用 C/C++コンパイラの pragma は、RX ファミリ用 C/C++コンパイラと互換性がありません。

#pragma abs16
#pragma abs20
#pragma abs28
#pragma abs32
#pragma regsave
#pragma noregsave
#pragma noregalloc
#pragma ifunc
#pragma tbr
#pragma global_register
#pragma gbr_base
#pragma gbr_base1
#pragma align4

これらの pragma を RX で使用している場合、コンパイル時に、以下のワーニングメッセージが出力されます。

W0520161:Unrecognized #pragma

又、割り込み関数を宣言する #pragma interrupt は、書式形式が異なります。RX ファミリ用 C/C++コンパイラの仕様に合わせて、適切に変更してください。

[SuperH]

#pragma interrupt [(]<関数名>[(割り込み仕様)][,…][)]

表　SuperH の割り込み仕様一覧

項目	形式
スタック切り替え指定	sp=<アドレス>
トラップ命令リターン指定	tn=<トラップベクタ番号>
レジスタバンク指定	resbank
レジスタバンク切り替え指定	sr_rts
RTS 命令リターン指定	rts

[RX]

#pragma interrupt [(]<関数名>[(＜割り込み仕様＞[,…])][,…][)]

表　RX の割り込み仕様一覧

項目	形式
ベクタテーブル指定	vect=<ベクタ番号>
高速割り込み指定	fint
割り込み関数レジスタ制限指定	save
多重割り込み許可指定	enable

4.3　組み込み関数

SuperH ファミリ用 C/C++コンパイラの組込関数のほとんどは、RX ファミリ用 C/C++コンパイラと互換性がありません。これらの組込関数を適切に除去するか、類似した機能をもつ RX ファミリ用 C/C++コンパイラの組込関数を使用してください。なお、DSP の組込関数は、RX では使用できません。

以下に SuperH と RX で類似した組み込み関数の対応一覧を示します。

表　組み込み関数対応一覧

SuperH	RX	機能
nop	nop	NOP 命令
swapb, swapw, end_cnvl	revl, revw	並べ替え
macw, macwl, macl, macll	rmpab, rmpaw, rmpal	積和演算
rotl, rotr, rotcl, rotcr	rotl, rotr, rolc, rorc	回転

組み込み関数を使用する場合には、必ず<machine.h> をインクルードしてください。RX では、<umachine.h>、<smachine.h> は、使用できません。

ホームページとサポート窓口

ルネサス エレクトロニクスホームページ
http://www.renesas.com/

お問合せ先
http://www.renesas.com/contact/

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.04.20	-	初版発行
2.00	2017.04.20	-	移行先を CS+,CC-RXV2 に変更

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 未使用端子の処理
   【注意】未使用端子は、本文の「未使用端子の処理」に従って処理してください。
   CMOS製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI周辺のノイズが印加され、LSI内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。未使用端子は、本文「未使用端子の処理」で説明する指示に従い処理してください。
2. 電源投入時の処置
   【注意】電源投入時は，製品の状態は不定です。
   電源投入時には、LSIの内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。
   外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。
   同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. リザーブアドレス（予約領域）のアクセス禁止
   【注意】リザーブアドレス（予約領域）のアクセスを禁止します。
   アドレス領域には、将来の機能拡張用に割り付けられているリザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
4. クロックについて
   【注意】リセット時は、クロックが安定した後、リセットを解除してください。
   プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。
   リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
5. 製品間の相違について
   【注意】型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。
   同じグループのマイコンでも型名が違うと、内部ROM、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ輻射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器・システムの設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因して、お客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
2. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
3. 本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害に関し、当社は、何らの責任を負うものではありません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を改造、改変、複製等しないでください。かかる改造、改変、複製等により生じた損害に関し、当社は、一切その責任を負いません。
5. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、
   各品質水準は、以下に示す用途に製品が使用されることを意図しております。
   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、
   家電、工作機械、パーソナル機器、産業用ロボット等
   高品質水準：輸送機器（自動車、電車、船舶等）、交通用信号機器、
   防災・防犯装置、各種安全装置等
   当社製品は、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（原子力制御システム、軍事機器等）に使用されることを意図しておらず、使用することはできません。たとえ、意図しない用途に当社製品を使用したことによりお客様または第三者に損害が生じても、当社は一切その責任を負いません。なお、ご不明点がある場合は、当社営業にお問い合わせください。
6. 当社製品をご使用の際は、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他の保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
8. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
9. 本資料に記載されている当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。また、当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途に使用しないでください。当社製品または技術を輸出する場合は、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。
10. お客様の転売等により、本ご注意書き記載の諸条件に抵触して当社製品が使用され、その使用から損害が生じた場合、当社は何らの責任も負わず、お客様にてご負担して頂きますのでご了承ください。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。
注2. 本資料において使用されている「当社製品」とは、注１において定義された当社の開発、製造製品をいいます。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口

http://www.renesas.com

※営業お問合せ窓口の住所は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス株式会社　〒135-0061　東京都江東区豊洲3-2-24（豊洲フォレシア）

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：https://www.renesas.com/contact/

Colophon 4.1